Accelerating HMMER on FPGA using Parallel Prefixes and Reductions

Naeem Abbas, Steven Derrien, Sanjay Rajopadhye, Patrice Quinton

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Accelerating HMMER on FPGA using Parallel Prefixes and Reductions

Naeem Abbas — Steven Derrien — Sanjay RajoPadhye — Patrice Quinton

N° 7370

August 2010

Architecture and Compiling

ISSN 0249-6399 ISRN INRIA/RR--7370--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
RENNES - BRETAGNE ATLANTIQUE

Accelerating HMMER on FPGA using Parallel Prefixes and Reductions

Naeem Abbas *, Steven Derrien * , Sanjay RajoPadhye * , Patrice Quinton *

Theme : Architecture and Compiling
Algorithmics, Programming, Software and Architecture
Équipes-Projets BioWic

Rapport de recherche n° 7370 — August 2010 — 24 pages

Abstract: HMMER is a widely used tool in bioinformatics, based on Profile Hidden Markov Models. The computation kernels of HMMER i.e. MSV and P7Viterbi are very compute intensive and data dependencies restrict to sequential execution. In this paper, we propose an original parallelization scheme for HMMER by rewriting their mathematical formulation, to expose the hidden potential parallelization opportunities. Our parallelization scheme targets FPGA technology, and our architecture can achieve 10 times speedup compared with that of latest HMMER3 SSE version, while not compromising on sensitivity of original algorithm.

Key-words: No keywords

* This work is supported by the French ANR BioWIC (ANR-08-SEGI-005).

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex
Téléphone : +33 2 99 84 71 00 — Télécopie : +33 2 99 84 71 71

Accelerating HMMER on FPGA using Parallel Prefixes and Reductions

Résumé : HMMER est un outil basé sur la notion profils à base modèles de Markov cachés, qui est très largement utilisé en bio-informatique. Les parties critiques de lâalgorithme (fonctions MSV et P7Viterbi) utilisées dans HMMER sont très consommatrices en temps de calcul et réputées très difficiles à paralléliser. Dans cet article, nous proposons un schéma de parallélisation original pour HMMER, basé sur une reformulation mathématique de lâalgorithme qui permet de découvrir de nouvelles possibilités de parallélisation bien adaptées à des implantations matérielles dédiées. Nous avons implanté cette approche sur un accélérateur FPGA et avons mesuré des gains en performance supérieurs à 10 par rapport à lâimplémentation logicielle de HMMER3, laquelle exploite pourtant déjà de manière extrêmement efficace les extensions SIMD des processeurs x86.

Mots-clés : Pas de motclef

Contents

1 Introduction

Profile Hidden Markov Models (HMMs) constitute an important computational tool extensively used in analyzing biological sequences, in particular for searching for statistically over-represented sub-sequences. One of the most commonly used program for HMM analysis is the open source software suite HMMER, developed at Washington University, St. Louis by Sean Eddy [5].

HMMER involves very computationally demanding algorithms and account for a large amount of time spent in biological sequence analysis. As a consequence, a great deal of effort has been put to improve its software performance through both fine grain (SIMD extension such as ALTIVEC and SSE extensions), coarse grain parallelisation (using MPI or multi-threads) and GPUs. Many authors have also investigated dedicated parallel hardware implementations, notably on FPGAs.

As currently defined and programmed, the latest version of HMMER spends most of its time in two kernel functions (called MSV and P7Viterbi). These two kernels contain loop carried dependencies (caused by the feedback path from the end to the beginning of model (the edges from M_5 through L to M_0 in Fig.1) which restricts any kind of parallelism.

We propose a technique to rewrite the computation in such a way that both kernels become very amenable to parallel implementation, while keeping all the original dependencies into account.

This research work makes two contributions. First, we describe how the original dynamic programming equations of MSV and P7Viterbi can be rewritten so as to develop the new algorithm that admits a scalable parallelization scheme at a price of a moderate increase in the algorithm computational volume. Then we propose several strategies for efficiently implementing this improved algorithm[1] on a FPGA-based High Performance Computing platform and discuss the performance that we obtained.

This document is organized as follows. Section 2 provides a detailed overview of previous work on parallelization of the HMMER software. Section 3 and 4 explain the approach we followed to rewrite the computations involved. Section 5 and 6 respectively focus on the FPGA implementation and on experimental results. Conclusion and future work directions are drawn in Section 7.

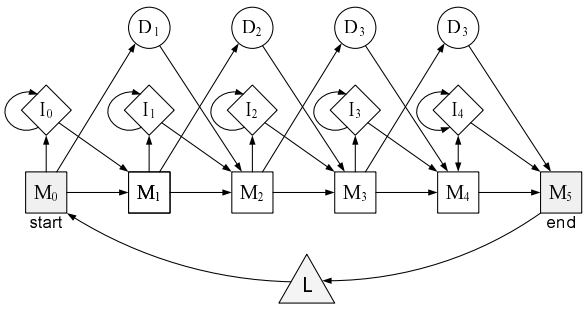


Figure 1: Structure of a Plan7 HMM

[1] In this work we are interested only in the hmmsearch software, as hmmpfam is known to be I/O bound

2 Related work

Because of its widespread use and of its heavy computing requirements, HMMER has received a lot of attention from the high performance computing community, with several implementations either for standard parallel machines or more heterogeneous architectures [16, 17]. In the following we will focus on hardware implementation targeting ASIC or FPGA technology.

2.1 Early HMMER implementations

Early proposals [11, 12] of hardware accelerator for profile based similarity search considered an (over) simplified version of the algorithm in which the feed-back loop is ignored as such a simplification happen to have a relatively limited impact on the actual Quality of Results of the algorithm.

The first hardware implementation of the exact algorithm was done by Oliver et al [13]. In their work, they reconsidered the problem by taking advantage of the fact that in hmmsearch, multiple independent instances of the Viterbi algorithm are to be executed during each call. They used this fact to model the problem as a triple nested loop (instead of a simple double nested loop) in which the most outer loop is parallel and proposed a SIMD architecture for execution the algorithm.

However, one of the issues with their parallelization scheme is that all processing elements need to concurrently access the (large) transition cost look-up tables. They addressed this problem by observing that all the processing element would only access a small subset of the table (24 locations, that is the number of bases in the Amino Acid alphabet). They therefore propose a vectorized memory with a 24-element wide data bus, in which each PE selects its cost value using a 24 to 1 multiplexer. Of course this approach suffers form severe scalability issues, which makes it impractical for massively parallel implementations.

In the mean time, another approach was proposed by Derrien and Quinton [4]. It also uses the fact that many instances of the Viterbi algorithm can be processed in parallel, however the parallelization scheme (based on polyhedral space-time transformation) is more sophisticated than that of Oliver et al. [13] and allows one to derive a flexible architectural skeleton which does not need access to a shared memory for calculating the transition costs. The proposed approach also easily handles resource constraints by controlling the number of processing elements in the architecture, and allows for precise tuning of the datapath pipeline. Although the approach does not suffer from previous shortcoming, its scalability is still somewhat limited, as the local storage requirements of the hardware implementation can be prohibitive. For example a 64-element processing array with a 6 stage pipelined datapath, would need more than 500 FPGA embedded memory blocks.

2.2 Speculative execution of HMMER Viterbi algorithm

More recently, an approach for hardware acceleration based on speculative execution was proposed by Takagi et al. [15] and Sun et al. [14]. Their idea is to take advantage of some property of the max operation, so as speculatively ignore the dependency over variable $X[i,k]$ since it very seldom contributes to

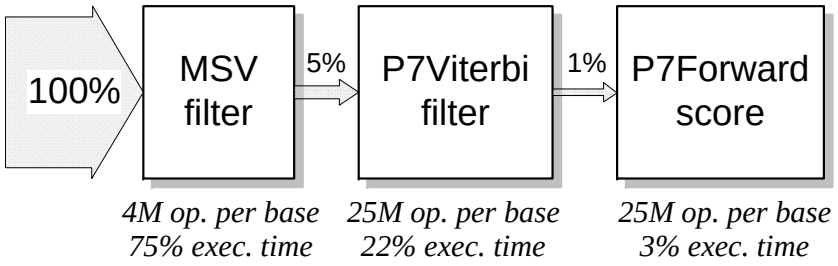


Figure 2: HMMER3 execution pipeline, with profiling data

the actual computation of $M[i+1,k]$, $D[i+1,k]$ and $I[i+1,k]$. This then results in a feedback-loop free algorithm, like [11]-[12], which is very easy to parallelize.

However, whenever it is observed that the actual value of $X[i]$ would have contributed to the actual value of $M[i+1,0]$, all computations related to columns i' such as $i' > i$ are discarded (flushed) and the computation must be re-executed so as to enforce the original algorithm dependencies. To do so Takagi et al. propose a misspeculation detection mechanism which stores in a buffer the values of M, D and I computed at the beginning of the new column (and their inputs) until the actual value of X is available (that is M_{prof} cycles later). The true values of M, D and I are then recomputed, and if they differ from the previous one, it means that the speculation was wrong, and that the previous results must be discarded.

One of the main issues with such an approach is the probability and cost of a misprediction. In this solution, whenever a misprediction occurs, the architecture has been running useless calculations during M_{prof} cycles. Assuming a misspeculation probability p, the execution overhead for a sequence of S amino acid bases can then be written as :

$$e = \frac{S + M_{prof}}{S + M_{prof} + pSM_{prof}}$$

As noticed by Takagi et al, the average observed value for p is 0.0001, which lead to an efficiency that vary between 94% and 99% depending on the depth of the speculation. It can also be observed that overhead is more important for an architecture exhibiting a large level of parallelism (the *depth* of the speculation being deeper), and for long sequences matched against small profiles, for which the probability of observing a repetition is cumulative with the sequence size. As an example Takagi et al. report cases where HMM profile characteristics lead to a poor efficiency (performance degradation by 85 %).

2.3 HMMER3 and The Multi ungapped Segment heuristic

The new version of HMMER, which is available for use now, is a radical rewrite of the original algorithm, with a clear emphasis on performance. The most noticeable difference in this new version lies in a new filtering heuristic (called *Multi ungapped Segment Viterbi*) which serves as a prefiltering step, and is executed before the standard P7Viterbi in the HMMER pipeline as illustrated

in Figure 2. This algorithmic modification alone helps improve performance by a factor of 10, and its algorithm is outlined below :

$$M[i,k] = \text{MSC}[k] + \max(M[i-1,k-1], X[i-1]) \quad (1)$$

$$X[i] = f_X(\max_k(M[i,k]) \quad (2)$$

It is to note that the algorithm still retains the feedback loop which restricts to compute entire M_i before start computing M_{i+1}, as X_i is being broadcasted to M_{i+1}. But as apparent from Eq. (1), M does not depend on I and D, which gives opportunity for faster computation.

HMMER	globin.hmm, M=143	Pkinase.hmm, M=255	rrm.hmm, M=77	fn3.hmm, M=84
V2	≈ 0.03	≈ 0.03	≈ 0.03	≈ 0.03
V3-noSSE	0.3	0.37	0.3	0.26
V3-SSE	5.2	7.16	2.83	2.65

Table 1: Performance in GCUP for Pfam-B.fasta

In addition to this filtering step, both P7Viterbi and MSV algorithm have also been redesigned so as to operate on short wordlengths (8 bits for MSV and 16 bits for P7Viterbi) so as to fully benefit from the SIMD extensions (SSE,Altivec) available on all Intel/AMD CPUs. As shown in table 1, the combination of the MSV pre-filtering stage with SIMD has a huge impact of the overall software performance, which is improved by a factor of more than 100, and which actually makes most previous FPGA implementations acceleration slower than any recent Dual-core CPU machine, as shown by Table 2.

	Min GCUPS	Max GCUPS
Takagi [15]	0.78	7.38
Y Sun [14]	0.28	3.2
T. Oliver [13]		5.3

Table 2: Reported average performance for previous FPGA implementation of HMMER2

2.4 Accelerating the complete HMMER3 pipeline

As shown in figure 2, because the MSV algorithm is used as a prefiltering step, the P7Viterbi algorithm also significantly contributes to the execution time. In other words, significantly improving global performance cannot be done by only accelerating the MSV kernel alone, as a consequence there is still a need for efficiently accelerating the P7Viterbi algorithm.

In the following section we propose to rewrite both MSV and P7Viterbi algorithms to make them amenable to hardware acceleration. We do so by using a simple reformulation of MSV equations to expose reductions operations, and by using an adaptation of the technique proposed by Gautam and Rajopadhye [6] to detect scans and prefix computations in P7Viterbi. This exposes a new level of parallelism in both algorithms that was previously unknown.

3 Rewriting the MSV Kernel

As mentioned earlier, the main computation in the MSV kernel is a dynamic programming algorithm that follows the standard algorithmic technique of filling up one data tables (called `M[i,k]` in this paper with `i` as the column index, and `k` as the row index) together with some other auxiliary variables. The values of the table entries are determined from previously computed entries (with appropriate initializations) using the following formulas.

$$M[i,k] = \text{MSC}[k] + \max(M[i-1,k-1], X_b[i-1]) \tag{3}$$

$$X_e[i] = \max(\max_k(M[i,k])) \tag{4}$$

Where X_b is computed as:

$$X_b[i] = \max(X_n[i-1] + \text{tloop}, X_e[i], X_e[i-1]) \tag{5}$$

It can be observed that the computation of M has a diagonal dependency for column M_{i-1} and X_b, where X_b depends on all value of M_{i-1} i.e. we can not start computation for column M_i, until we have computed column M_{i-1}, which gives column-wise sequential execution to the algorithm.

On the other hand, all values of a given column M_i can be computed in parallel, and since the computation of X_e consists of a max reduction operation, It can be realized using a max tree computation as shown in Fig.3

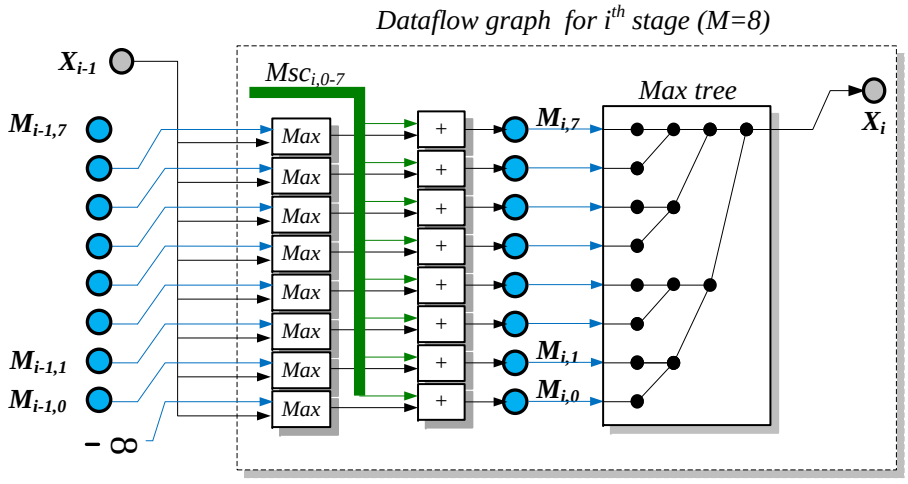


Figure 3: Dataflow dependencies for one stage of the MSV filter ($M = 8$) algorithm after rewriting

4 Rewriting the P7Viterbi Kernel

As shown in previous section, it is easy to rewrite the MSV algorithm recurrence equations so as to expose parallelism in the form of a simple max-reduction operation.

In this Section, we show how it is also possible to use a similar (but more complex) rewrite on the P7Viterbi kernel. Here again, the goal is to get rid of the current inherent sequential behavior caused by the so-called *feed-back loop*. To do so we replace the accumulation along the j index for one of the variables by a prefix-scan operation and replace the feed-back loop by a simple max-reduction operation. This transformation leads to a modified dependence graph which is much better suited for a parallel hardware implementation.

4.1 The P7Viterbi algorithm

The P7Viterbi algorithm is somewhat similar to MSV, except that it fills three data tables (called `M[i,k]`, `I[i,k]` and `D[i,k]`) together with some other auxiliary variables. The values of these tables entries are determined using the following formulas, where the column index, i is written as a subscript in order to focus on the important dependences.

$$M_i[k] = f_M(M_{i-1}[k-1], I_{i-1}[k-1], D_{i-1}[k-1], X_{i-1}) \quad (6)$$

$$I_i[k] = f_I(M_{i-1}[k], I_{i-1}[k]) \quad (7)$$

$$D_i[k] = f_D(M_i[k-1], D_i[k-1]) \quad (8)$$

$$X_i = f_X(\max_k(M_i[k] + E[k])) \quad (9)$$

Here, `X[i]` is an auxiliary computed variable, `E[k]` is an input, and f_M, f_I, f_D and f_X are some simple functions. The key observation concerning these formulas is that

- there is a chain of dependences in the increasing order of k in computing the values of D in any column;
- to compute the X for any column, we need *all* the values of M of that column, each of which needs a D from the previous column; and
- the value of X of a column is needed to compute *any* M in the next column.

Because of this, there seems to be an inherent sequentially to the algorithm, as noted by all previous work on this problem.

4.2 Finding reductions

We now develop an alternate formulation of the equations so that there is no such chain of dependences, thus enabling scalable parallelization of the computations on a hardware accelerator.

More specifically, we show that the equation computing D can be replaced by a different equation in which such dependences either do not exist, or can be broken through well-known techniques. For our purposes, we shall focus on the function f_D, which is defined more precisely as follows:

$$D_i[k] = \begin{cases} k = 1 & : \quad M_{i-1}[0] + A'[0] \\ k > 1 & : \quad \max(D_i[k-1] + B[k], M_{i-1}[k] + A'[k]) \end{cases} \tag{10}$$

where A' and B are appropriate inputs. From the point of view of the computation in the i-th column, we can consider the values for the previous column (viz., the M_{i-1} terms above) as inputs, and so the equation can be further abstracted as follows:

$$D_i[k] = \begin{cases} k = 1 & : \quad a_0 \\ k > 1 & : \quad \max(D_i[k-1] + b_{k-1},\ a_{k-1}) \end{cases} \tag{11}$$

Now, if B is zero, the equation is a simple *scan computation* (also called *prefix computations*) $D_i[k] = \max_{i=1}^{k} a_i$.

It is well known how to efficiently and scalably parallelize such scan computations [10]. However, if $B \neq 0$, the solution is not at all obvious. We show below how to obtain a scan-like structure for this case. If we expand out the individual terms, we see that:

$$\begin{aligned}
D[1] &= a_0 \\
D[2] &= \max(a_0 + b_1, a_1) \\
D[3] &= \max(\max(a_0 + b_1, a_1) + b_2, a_2)) \\
&= \max(a_0 + b_1 + b_2, a_1 + b_2, a_2) \\
D[4] &= \max(a_0 + b_1 + b_2 + b_3, a_1 + b_2 + b_3, a_2 + b_3, a_3) \\
&\vdots \\
D[k] &= \max(a_0 + b_1 + b_2 + b_3 \ldots b_{k-1}, a_1 + b_2 + b_3 \ldots b_{k-1}, \\
&\quad\ a_2 + b_3 \ldots b_{k-1}, \ldots a_{k-2} + b_{k-1}, a_{k-1})
\end{aligned}$$

The last term can be written more visually as

$$D[k] = \max \left(a_{k-1}, \max \left(\left(\begin{array}{r} b_1{+}b_2{+}b_3{+}\ldots{+}b_{k-1} \\ b_2{+}b_3{+}\ldots{+}b_{k-1} \\ b_3{+}\ldots{+}b_{k-1} \\ \vdots \\ b_{k-1} \end{array} \right) + \left(\begin{array}{c} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{k-2} \end{array} \right) \right) \right)$$

Writing this more compactly, we proceed as follows.

$$\begin{aligned}
D[k] &= \max \left(a_{k-1}, \max_{j=1}^{k-1} \left(a_{j-1} + \sum_{i=j}^{k-1} b_i \right) \right) \\
&= \max \left(a_{k-1}, \max_{j=1}^{k-1} \left(a_{j-1} + \sum_{i=j}^{k-1} b_i + \sum_{i=1}^{j-1} b_i - \sum_{i=1}^{j-1} b_i \right) \right) \\
&= \max \left(a_{k-1}, \left(\sum_{i=1}^{k-1} b_i + \max_{j=1}^{k-1} \left(a_{j-1} - \sum_{i=1}^{j-1} b_i \right) \right) \right)
\end{aligned}$$

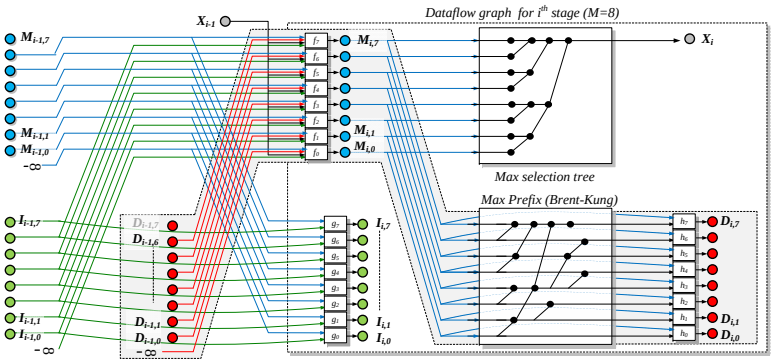


Figure 4: Dataflow dependencies for one stage of the P7Viterbi ($M = 8$) algorithm after rewriting

Now note that $\sum_{i=1}^{j-1} b_i$ (say, b'_{j-1}) is a scan of the B input, so

$$\begin{aligned} D[k] &= \max\left(a_{k-1}, b'_{k-1} + \max_{j=1}^{i-1}(a_{j-1} - b'_{j-1})\right) \\ &= \max\left(a_{k-1}, b'_{k-1} + \max_{j=1}^{i-1} a'_{j-1}\right) \end{aligned} \tag{12}$$

where $a'_j = a_j - b'_j$ is the element wise difference of a and b'.

Now, the inner term is max-scan of the a' vector. Hence the $D[k]$ as specified by Eqn. 12 can be computed in parallel using the following steps.

1. Compute, B' the sum-prefix of the array B. Note that in the Viterbi algorithm, this needs to be done only once since B is an input.
2. Compute C, the element wise subtraction of this from A.
3. Perform a max-prefix on C. This can be parallelized perfectly and scalably.
4. Add B' element wise to this and compare (again element wise) the result with the A input, retaining the larger one. This yields D, the desired answer.

4.3 Impact of the data-dependence graph

To help the reader understanding the benefits of this rewriting transformation, we provide in Fig. 4 an illustration of the data dependence flow in the rewritten algorithm for a small problem size (profile size $M = 8$). In this dataflow graph, functions $f_{i,k}$,$g_{i,k}$ and $h_{i,k}$ are defined as follows :

$$\begin{aligned} f_M(w,x,y,z) &= \max_4(w + bsc_k, x + TMM_k, y + TDM_k, \\ &\quad z + TIM_k) + msc_k[dsq_i] \\ g_I(x,y,z) &= \max_2(x + TII_k, y + TMI_k) + isc_k[dsq_i] \\ h_D(x,y) &= \max_2(x + TMD_k, y + y) \end{aligned}$$

In these expressions max and *sum* correspond to saturated (w.r.t to $-\infty$) *max* and *sum* operations. It can be observed, that there is no longer a chain of dependency along the vertical axis in the data-flow graph, and that the longest path (i.e. critical path) is now set by the depths of the parallel *max-tree* and the parallel *max-prefix* blocks, which is $O(log_2(M))$. Another consequence is that update operations for $M_{i,k}$, $I_{i,k}$ and $D_{i,k}$ can be executed in parallel for all values of k in the domain $0 \leq k \leq M$.

5 Mapping the computations to hardware

Even though the rewritten version of both MSV an P7Viterbi algorithms exhibits a significant amount of *hidden* parallelism, deriving a efficient architecture from the modified dataflow graph is not straightforward.

In this section we address the different challenges involved in this architectural mapping. We first start by discussing efficient hardware implementation of parallel prefix operations as needed by P7Viterbi, and present two transformations (namely C-Slow and tiling) that we use to improve the architecture efficiency.

5.1 Implementing the max-prefix operator

As mentioned in Section 4.B, step 3, the rewritten version of the P7Viterbi algorithm exhibits a max-prefix pattern. Prefix computation is a very general class of computations which can be formally defined as follows : given an input vector x_i with $0 \leq i < N$ we define its $\oplus$-prefix vector y_i as :

$$y_i = \bigoplus_{k=0}^{i} x_k = x_0 \oplus x_1 \oplus \ldots \oplus x_i$$

Where $\oplus$ is a binary operator with associativity (and possibly commutativity, see [8] for a more detailed definition). Because binary adders fall into this category and since adders form one of the most important building blocks in digital circuits, there is a wealth of research going back almost 50 years dealing with fast (aka parallel) implementations of prefix adders [10, 3, 9, 7] using various interconnection networks topologies.

One of the most important aspects of these network topologies is that they allow the designer to explore the trade-off between speed (i.e. critical path of the resulting circuit), area (number of operators used to implement the prefix operation), and others metrics such as fan-out or wiring length.

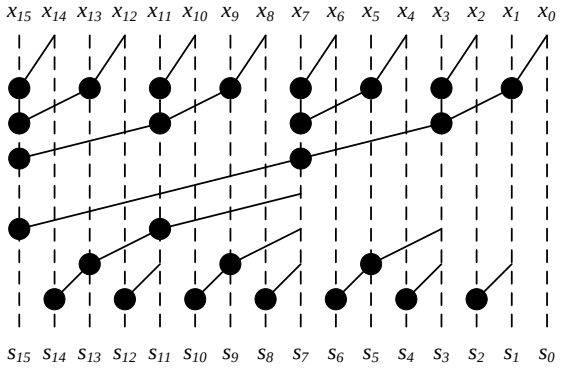
For example, Figure 5.a shows a Brent-Kung[3] network that computes the prefix in $2(\log_2 N - 1)$ stages with $2(N-1) - \log_2 N$ operators. Similarly, Figure 5.b shows a Ladner-Fischer network which implements a faster circuit ($\log_2 N$ stages) at a price of an increase in area ($\frac{N}{2} \log_2 N$ operators).

Although a survey of all existing techniques and network topologies is clearly out of the scope of this paper, we provide in table 3 a short summary of the characteristics of the most commonly used algorithms. Note that most of these algorithmic explorations were in a context where the operator was extremely fine grain—just a few Boolean gates, as in a half- or full-adder.

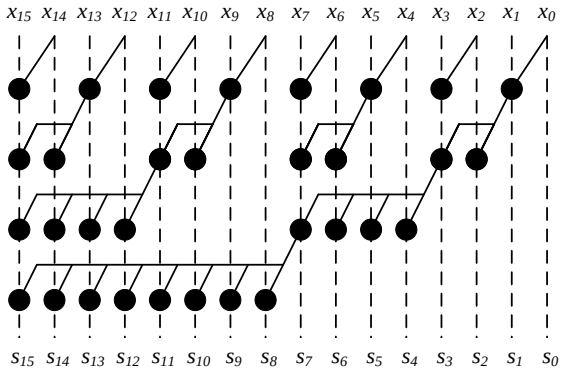
Method	Delay	Cost
Ladner-Fischer[10]	$\log_2 N$	$\frac{N}{2} \log_2 N$
Kogge-Stone [9]	$\log_2 N$	$N \log_2 N - N + 1$
Brent-Kung [3]	$2 \log_2 N - 2$	$2N - 2 - \log_2 N$

Table 3: Characteristics of various parallel-prefix networks

Despite the fact that our computation scheme is based on the same prefix patterns as binary adders, our situation differs in two ways :



(a) Brent Kung



(b) Ladner-Fischer

Figure 5: Two examples of parallel parallel prefix implementation for $N = 16$

- The basic operation is not a bit-level (i.e full-adder) but a more complex word level operation (namely max).
- The size of the prefix can be very large (up to 256 input elements) which poses scalability issues in terms of routing.

To the best of our knowledge there has been no systematic study of FPGA implementation of prefix computations. One reason is that the typical use of such circuits would be in adders, where high-speed carry circuits are already provided by FPGA vendors, and there are few applications that need coarse-grain, word-level operators. For the HMMER application we implemented a number of the max-prefix as well as max-reduce architectures. The performance comparisons are reported later in Section VI.

5.2 Mapping dataflow as a combinational datapath

It can be easily seen from Fig.4 and Fig.3, that in both MSV and P7Viterbi, it is not possible to pipeline the execution of consecutive stages —all the results of the i^{th} stage are needed before ANY value in the (i+1)th stage can be computed.

As a consequence, and in spite of the fact that we replaced in both cases the initial chain of dependence of $O(M)$ operations by a chain of $O(\log_2(M))$, the possibly large values of M may induce a large critical path, which will in turn lead to a poor clock frequency.

5.3 Mapping dataflow as a C-slowed pipelined datapath

One solution to improve the through-put of the hardware implementation is to use the same approach as used by Derrien et al [4], and also Oliver et al. [13] by applying a C-slow transformation on the generated datapath so as to interleave several execution of the dataflow graph (that is several independent instances of the MSV/P7Viterbi algorithm).

Another way to view this transformation is to consider that we add an additional outer loop iterating over independent instances of the algorithm, and then perform a loop interchange so as to move this parallel outer loop to the innermost level and implementing the multiple independent instances on pipelined hardware parallelism.

Using this trick, and assuming the interleaving of S independent instances, the i^{th} stage now only depends on stage that was executed $i - S$ stages before. This extra delay can then be used to pipeline the stage execution, as depicted in Figure 6a.

This of course has some additional memory cost (we must replicate all registers/memories in the architecture), but because the critical path remains in $O(\log_2 M)$, we only need a reasonably small C-slow factor to achieve the maximum through put (as compared to $S \approx O(M)$ in the approach of Derrien and al.).

5.4 Managing resource constraints through tiling

Both MSV and P7Viterbi dataflow graphs sizes scale linearly[1] with the target HMM profile size M. For large values of profile sizes (e.g., greater than 100) the straightforward mapping of the complete dataflow graph to a hardware datapath quickly becomes out of reach of most FPGA platforms.

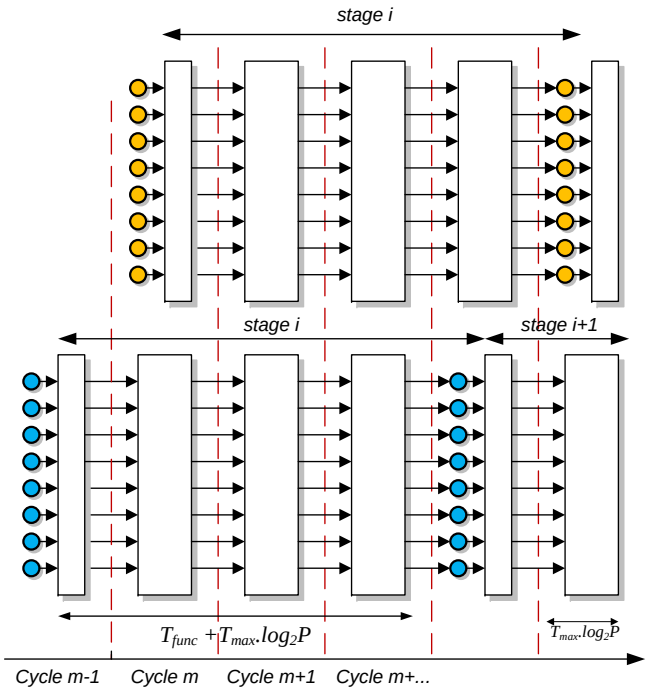
However, since the computational pattern of both algorithms exhibits a lot of regularity, it is possible to apply a simple *tiling* transformation, which tiles each dataflow of size M into P partitions, each of them calculating M/P consecutive values of the current column. This transformation, and its impact on the scheduling of computation is depicted in Figure 6b. In the case of MSV, the partitioned datapath should implement a M/P reduction max operator, whereas in the case of P7Viterbi, we need a M/P max prefix operation.

As a summary, the characteristics of various design space points that we explored are listed in Table 4.

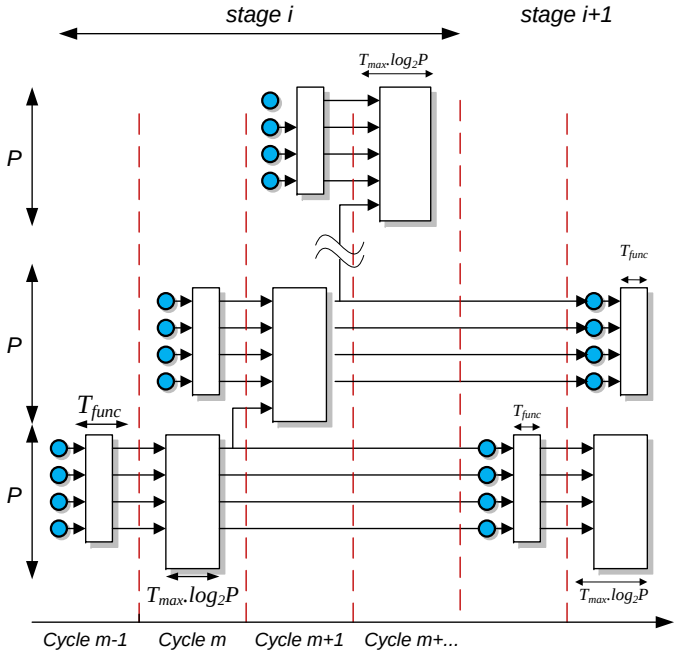
5.5 Accelerating the full HMMER execution pipeline

As mentioned in subsection 2.4, improving global performance requires that both MSV and P7Viterbi are accelerated in hardware. This can be done very easily by streaming the output of MSV to the input of the P7Viterbi, so as to

[1] The scaling is linear for the Brent-Kung architecture that we implemented. For the Ladner-Fischer architecture, the resource usage grown as $n \log n$



(a) C-slow



(b) Tiled Mapping

Figure 6: C-slow and Tiled Dataflow graphs

Method	Area	T_{clk}	Through-put
Combinational	$O(M)$	$O(\log_2 M)$	$O(\frac{M}{\log_2 M})$
Tiled	$O(M/P)$	$O(\log_2 \frac{M}{P})$	$O(\frac{M}{\log_2 \frac{M}{P}})$
C-slow	$O(M)$	$O(1)$	$O(M)$
Tiled + C-slow	$O(M/P)$	$O(1)$	$O(M/P)$

Table 4: A summary of the different architectural solutions, along with their space-time characteristics

map the complete HMMER3 pipeline to hardware. Special care must however be taken of the C-Slow factor for both accelerators, which must be the same so as to avoid a complex data reorganization engine between the two accelerators.

In addition, depending on available resource, it is even possible to instantiate several HMMER3 pipelines in parallel, as illustrated in Figure 7.

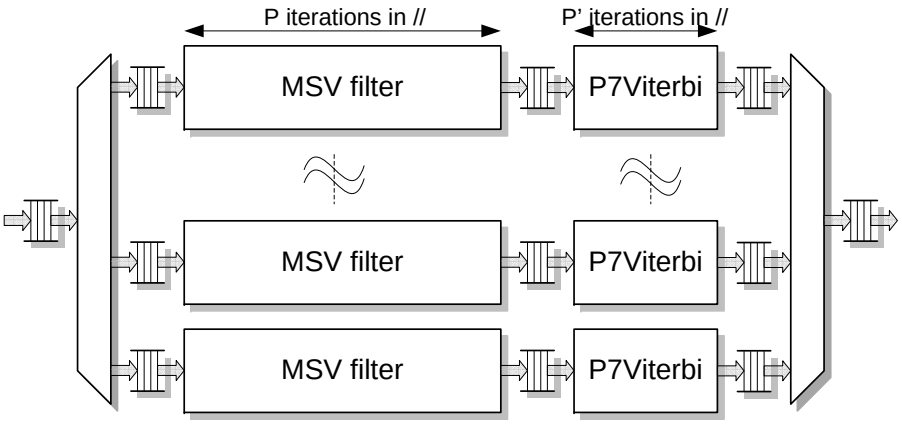


Figure 7: System level view of a complete HMMER3 hardware accelerator

However, in order to optimize hardware resource usage, we must also ensure that the pipeline workload is well distributed among the hardware accelerators. Let us quantify the total algorithm execution time, written as T_{total} when the two task executions are pipelined, we have :

$$T_{\text{total}} = \max\left(T_{\text{msv}}, \alpha T_{\text{viterbi}}\right)$$

Where T_{MSV} and $T_{Viterbi}$ correspond to the average algorithm execution times, and where α is the filtering selectivity. Optimizing performance therefore means ensuring the raw performance (in GCUPS) of the P7Viterbi accelerator is able to sustain the filtered output of the MSV accelerator, that is, it's performance should be at least $1/20^{th}$ that of MSV.

Using this constraint we can then define a set of pipeline configurations, by choosing distinct tiling parameters (i.e. partition sizes) for P7Viterbi and MSV such that the level of parallelism exposed in MSV is at least $20\times$ that of P7Viterbi.

6 Experimental results

In this section we provide an in-depth quantitative analysis of our proposed architectures, and compare their performance with that of state of the art software implementation of HMMER3 on CPU using the SSE SIMD implementation.

Our target execution platform consists in a high-end FPGA accelerator from XtremeData (XD2000i-FSBFPGA) which has already been successfully used for implementing bioinformatics algorithms [2]. This platform contains two Stratix-III 260 FPGAs, high bandwidth local memory (8.5 GBytes/s) and a tight coupling to the host front side bus though Intel Quick Assist technology, providing sustained 2 GBytes/s bandwidth between the FPGA and the host main system memory.

Our design flow leverages on High-level synthesis through a commercial C to Hardware compiler (Impulse-C) combined with a semi automated source to source compiler [1] so as to ease architectural design space exploration.

The rest of the section is organized as follows, we first make a quantitative analysis of performance/area results for the MSV accelerator, then address the mapping of max prefix network implemented on FPGA along with P7Viterbi implementation results. Finally, we discuss system level performance and compare our performance of our approach with that of an hypothetic GPU implementation.

6.1 Area/performance results for the MSV filter

Table 5 summarizes area and performance results obtained for several MSV hardware accelerators using different values of M and S (the MSV accelerator doesn't need tiling as for all profile sizes, it fits in the FPGA). It can be observed, that even though we use a C to hardware high level synthesis tool, we are able to achieve remarkably high operating frequencies (up to 215 MHz). When compared[2] with that of table 1 results indicate speedup for a single accelerator varying between 3× to 6× depending on M.

M	C-Slow (S)	Logic Util.	M9K	MHz	GCUPS
64	7	10k / 5%	66 / 8%	215	14
128	8	19k / 9%	130 / 15%	201	25
256	9	37k / 19%	258 / 30%	175	45
512	10	69k / 34%	513/60%	160	81

Table 5: Performance and resource usage of our MSV hardware implementation

6.2 Area/performance results for max-prefix networks

As mentioned in Section 5, the P7Viterbi implementation uses a parallel max prefix scheme, for which many implementation scheme exists. As this computational pattern is at the core of the modified algorithm, we explored several alternative implementations so as to experimentally quantify their respective merits with respect to an FPGA implementation.

[2]This is an rough approximation, as we should also account for the time spent by the software in P7Viterti (20% of the total execution time)

Method	16	32	64	128
Area				
Brent-Kung	883	1879	3972	8030
Ladner-Fischer	512	2728	6266	14226
Kogge-Stone	1846	4637	11394	26786
F_{max}				
Brent-Kung	82	59.7	47.24	36.91
Ladner-Fischer	109	79.6	65.6	53.4
Kogge-Stone	102	76.8	61	46.8

Table 6: Speed/Area results for combinational parallel max prefix implementations on Stratix-II FPGA

The results are provided in table 6 show that for large values of M, so called *fast* implementations of parallel prefix such as Kogge-Stone or Ladner-Fischer provide only marginal speed improvements with respect to the Brent-Kung architecture. This can easily be explained by the long wires used in the two first two approaches, which make the routing much more challenging on an FPGA.

6.3 Area/performance results for the P7Viterbi filter

Table 7 summarizes area and performance results obtained for several P7Viterbi hardware accelerators using different values of M,S and P. It can be observed, that re-written P7Viterbi kernel can deliver quite promising performance with $\log_2(M)$ CSlow factor. By fitting multiple instances of P7Viterbi on board, It can alone(i.e. not using MSV filter) perform better than earlier implementation of HMMER2.

P	M	Logic Util.	M9K	MHz	GCUPS
8	64	5.8K / 2.8%	69 / 8%	126	1
16	64	10.1K / 4.9%	112 / 13%	119	1.9
8	128	6.8K / 3.3%	128 / 14.8%	124	0.99
16	256	14K / 6.9%	170 / 19.7%	117	1.87
32	256	28.7k / 14%	332 / 38%	112	3.6

Table 7: Performance and area for our P7Viterbi implementation

6.4 System level performance

So far, we have provided area/performance results only for standalone accelerator modules, which should at some point be integrated together in one or several complete HMMER3 computation pipelines.

Following the constraints on pipeline workload balancing, and given the resource available implementing the accelerator on the board, we derived a set of pipeline configuration depending on the target profile size M.

M	C-Slow (S)	P	R	Est. GCUPS	Through-put
64	7	8	5	69	1075 MB/s
128	8	16	2	51	402 MB/s
256	9	16	1	45	175 MB/s

Table 8: Pipeline configuration on a single StratixIII260

These configuration have parameters (C-Slow factor S, Tiling parameter P, numer of parralel pipeline R) chosen so as to maximise overall performance. The set of parameters for a given value of M in chosen as follows:

- First, we choose the C-slow factor S so as to enable the fine grain pipelining (i.e. at the operator level) of the MSV accelerator. The same value is then used for the pipelining the P7Viterbi accelerator.
- Second, we choose the Partition P size so that the P7Viterbi accelerator can sustain at least 1/20 of the MSV input through-put.
- Last, once we have all parameters defining a single HMMER3 pipeline, we try to pack as many of them as possible (maximize R) on the target FPGA platform so as to maximise parallelism.

Table 8 provide some of the pipeline configurations that we obtained through this approach. At the time of the submission of the paper, we are still finalizing the validation of complete systems with full pipelines. We expect to present more global performance results in the final version of this work, but we must emphasis that the performance currently provided in Table 8 are only estimates and not actual measurements. These estimates show that speedup of up to ??× could be achieved using only one of the two FPGA of the platform. They were obtained using the equation below, where f_{MSV} (resp. f_{P7V}) correspond to the MSV (resp. P7Viterbi) accelerator maximum clock frequency.

$$GCUPS = R(Mf_{MSV} + \frac{1}{20}M/Pf_{P7V})$$

Even though extrapolating system-level performance from P&R data is somewhat disputable, we are confident that our architecture performance will be close to our estimates, as we have practical evidence that the I/O requirements shown in the last columns of Table 8 are within the reach of our target FPGA platform, for which we have measured a sustained Host to FPGA through-put of more than 1.5 GByte/sec.

6.5 Discussion

One question raised by our results is whether our FPGA would actually perform faster than an equivalent GPU implementation. This is an important point as GPU offers more flexibility at a much lower cost than a typical HPC FPGA platform, even if their power consumption is very high. Sadly, there is currently no GPU version of HMMER3 available for comparing the two targets.

We however believe that, to the contrary of HMMER2, a GPU version of HMMER3 would only offer marginal performance improvements w.r.t the optimized SSE version. Indeed, previously reported GPU performance [16] improvements for HMMER2 were in the order of $15 - 35\times$ for a single GPU over the software implementation, this later one using 32 bit arithmetic and not taking advantage of Intel SIMD extensions.

When looking at HMMER3 performance results (given in Table 1), it turns out that the use of an optimized SSE software implementation alone brings up to $20\times$ performance improvement over the non SSE version, a speed-up somewhat similar than that of the GPUs implementation for HMMER2, and which is mostly due to the systematic use of sub-word parallelism. As GPUs do not have support for short integer sub-word parallelism, it is therefore very unlikely that they will do much better than SSE implementation.

7 Conclusion

In this document, we have proposed a hardware accelerator architecture for the new HMMER3 profile based search software. This architecture leverages on a rewriting of the algorithm to expose reduction and prefix scan computation patterns without modifying the semantic of the original algorithm. This rewriting permit a full and efficient parallelization of the two kernels on hardware, and is combined with an architectural design space exploration stage so as to determine the best performing architecture for a given HMMER profile size, by taking into account the amount of available hardware resource and the pipeline workload balance. Our first estimates shows that speedup between 10× could be obtained w.r.t to the state of the art software implementation. Our ongoing work is currently geared toward obtaining and analysing profiling data from real-life workloads. We Also considering to implement a complex filter in between MSV and P7Viterbi, which can feed multiple inputs from MSV to a single P7Viterbi. This will enable us to use hardware resources at best. We may obtain a speedup of 15× with such a filter implementation.

References

[1] The Gecos compiler infrastructure. http://gecos.gforge.inria.fr/doku.php.

[2] Jeffrey Allred, Jack Coyne, William Lynch, Vincent Natoli, Joseph Grecco, and Joel Morrissette. Smith-Waterman implementation on a FSB-FPGA module using the Intel Accelerator Abstraction Layer. In *IPDPS*, pages 1–4. IEEE, 2009.

[3] R.P. Brent and H.T. Kung. A regular layout for parallel adders. *Computers, IEEE Transactions on*, C-31(3):260 –264, march 1982.

[4] Steven Derrien and Patrice Quinton. Parallelizing HMMER for hardware acceleration on FPGAs. In *ASAP 2007, 18th IEEE International Conference on Application-specific Systems, Architectures and Processors*, July 2007.

[5] S. Eddy. Sequence Analysis Using Profile Hidden Markov Models. Technical report, Washington University at Saint Louis, 2004.

[6] Gautam and S. Rajopadhye. Simplifying reductions. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 30–41, New York, NY, USA, 2006. ACM.

[7] T. Han and D. A. Carlson. Fast area-efficient vlsi adders, 1987.

[8] S. Knowles. A family of adders. In *ARITH '99: Proceedings of the 14th IEEE Symposium on Computer Arithmetic*, page 30, Washington, DC, USA, 1999. IEEE Computer Society.

[9] Peter M. Kogge and Harold S. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Trans. Comput.*, 22(8):786–793, 1973.

[10] Richard E. Ladner and Michael J. Fischer. Parallel Prefix Computation. *Journal of ACM*, 27(4):831–838, 1980.

[11] R. P. Maddimsetty, J. Buhler, R. D. Chamberlain, M. A. Franklin, and Brandon Harris. Accelerator Design for Protein Sequence HMM Search. In *Proceedings of the ACM International Conference on Supercomputing*, Cairns, Australia, 2006. ACM.

[12] T. Oliver, B. Schmidt, Y. Jakop, and D. L. Maskell. Accelerating the Viterbi Algorithm for Profile Hidden Markov Models Using Reconfigurable Hardware. In *International Conference on Computational Science*, 2006.

[13] T. Oliver, L. Y. Yeow, and B. Schmidt. High Performance Database Searching with HMMer on FPGAs. In *HiCOMB 2007, Sixth IEEE International Workshop on High Performance Computational Biology*, march 2007.

[14] Yanteng Sun, Peng Li, Guochang Gu, Yuan Wen, Yuan Liu, , and Dong Liu. HMMER Acceleration Using Systolic Array Based Reconfigurable Architecture. In *IEEE International Workshop on High Performance Computational Biology*, 2009.

[15] Toyokazu Takagi and Tsutomu Maruyama. Accelerating HMMER Search Using FPGA. In *International Conference on Field Programmable Logic and Applications*, September 2009.

[16] John Paul Walters, Vidyananth Balu, Suryaprakash Kompalli, and Vipin Chaudhary. Evaluating the use of gpus in liver image segmentation and hmmer database searches. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

[17] Ben Wun, Jeremy Buhler, and Patrick Crowley. Exploiting Coarse-Grained Parallelism to Accelerate Protein Motif Finding with a Network Processor. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, 2005.

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399